Stdg Reference Manual

All function and structure definitions can be found in the "stdg.h" include file.
The library is currently only implemented for the C programming language,
but exists on both the Apple Macintosh and Microsoft Windows platforms.

Initialisation

SYNOPSIS

```
int         main(int argc, char **argv);
void        ginit(char *name, voidfn *adjust_menus, menu *menubar);
void        gexit(void);
void        gflush(void);

extern void (*gerror)(char *errstr);

extern bitmap *     screen;
extern font *       sys_font;
extern font *       fixed_font;
extern cursor *     current_cursor;
extern window *     active_window;
extern short        menu_item;
```

DESCRIPTION

The program begins in the *main* function, which must be defined as shown. The arguments and the return value will only have meaning on UNIX platforms.

The function *ginit* initialises the structures necessary to use the library's graphics interface. The **name** argument specifies the name of the application, which can be sent to its environment if necessary. The name is used on UNIX platforms to uniquely specify a resource directory where application resources are kept. The *adjust_menus* function is called just after the user clicks in the application's menubar, and just before any menus are displayed. It can thus ensure the menus correctly reflect the current state of the program.

The **menubar** argument is a NULL terminated array of menus, each **menu** being a NULL terminated array of **menuitems**. An application has only one menubar, which may be displayed at the top of the screen, or at the top of the first window the application creates depending on the environment the program is running in.

The *gexit* function disposes of the application's graphical resources, and should be called at the end of each program. It will close all of the application's windows as part of its actions.

Graphics operations on some platforms (such as X-Windows) may be buffered, and for those platforms calling *gflush* ensures all pending graphics requests are processed. On other platforms the function exists and does nothing. All event handling routines call *gflush*, so usually it will not be necessary to call it.

If the *gerror* function is not NULL, it will be called by the library whenever an internal error occurs. The default error function displays the error string **errstr** in a window, and after the user clicks with the mouse, the *gexit* function is called and the program ends. If *gerror* is NULL, no error will be raised and the programmer will have the opportunity of testing for the error condition by checking return values. The programmer can set *gerror* to be a custom error reporting function, as long as its calling interface

is the same.

Upon initialisation, the library sets **screen** to be the bitmap of the screen. The screen bitmap can be used to find the depth or size of the screen, but it is not guaranteed that drawing to the screen bitmap will work. The **sys_font** is the platform's normal system font and **fixed_font** is a fixed width font for use by the application. The **current_cursor** is guaranteed to point to the currently displayed program cursor. The **active_window** pointer will always point to the active application window, or be NULL if there is no such window. The variable **menu_item** is set by the library to be the array index of the last selected menu item.

Basic Structures

SYNOPSIS

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned int   uint;
typedef unsigned long  ulong;

struct    point {
          long      x;          /* horizontal co-ordinate */
          long      y;          /* vertical co-ordinate */
};

struct    rectangle {
          point     min;        /* top-left point inside rectangle */
          point     max;        /* bottom-right outside rectangle */
};
```

DESCRIPTION

A **point** is a location in a bitmap (see below), and is defined as:

```
struct point { long x; long y;};
```

The coordinate system has x increasing to the right and y increasing down.

A **rectangle** is a rectangular area in a bitmap.

```
struct rectangle { point min; point max; };
```

By definition, **min.x** <= **max.x** and **min.y** <= **max.y**. By convention, the right (maximum x) and bottom (maximum y) edges are excluded from the represented rectangle, so abutting rectangles have no points in common. Thus, **max** contains the coordinates of the first point beyond the rectangle.

Bitmaps

SYNOPSIS

```
struct      bitmap {
            rectangle   r;          /* rectangle in data area, local coords */
            short       depth;      /* depth = number of bits per pixel */
            uchar *     bits;       /* bitmap data */
};

bitmap *    new_bitmap(rectangle r, short depth);
bitmap *    get_bitmap(char *name, short depth);
void        del_bitmap(bitmap *b);
```

DESCRIPTION

A **bitmap** holds a rectangular image.

```
struct bitmap { rectangle r; short depth; uchar *bits; };
```

The rectangle **r** specifies the bitmap's size in pixels. A bitmap need not have a zero origin. There are **depth** contiguous bits for each pixel of the image; the bits form a binary number encoding each pixel's colour. **Bits** is a pointer used internally by the library to access the bitmap.

New_bitmap creates and returns a pointer to a new white-filled bitmap. The rectangle **r** is the pixel size of the bitmap and **depth** is the number of bits per pixel. Depths of 1, 2, 4, 8, 16 and 32 bits per pixel are supported on the Macintosh, while depths of 1 and 4 are supported by most VGA drivers for PC compatibles. If an error occurs, *gerror* will be called, or the function will return NULL if *gerror* is NULL.

Get_bitmap searches the application's resources for a bitmap of the required name and depth, and returns the bitmap or NULL if it cannot be found. *Del_bitmap* de-allocates the memory used by **b**. If any of the bitmap functions are passed a depth of zero, the screen's depth will be substituted. If a bitmap of the required depth cannot be located, one of a lesser depth may be returned instead.

Windows

SYNOPSIS

```
typedef void (winfn)(window *);   /* window function */

struct      window {
            bitmap *    b;          /* bitmap for the window */
            rectangle   r;          /* window rectangle on screen */
            ulong       flags;      /* window appearance bit array */
            short       kind;       /* user data can be stored here */
            void *      data;       /* user data can be stored here */
            winfn *     close;      /* called before window is closed */
            winfn *     resize;     /* called after window is resized */
            winfn *     redraw;     /* called when window must be redrawn */
};

#define     Simple                  0x0000
#define     Visible                 0x0001
#define     Buffered                0x0002
#define     Titlebar                0x0004
#define     Closebox                0x0008
#define     Maximize                0x0010
#define     Resize                  0x0020
#define     Attentive               0x0040
#define     DoubleClicks            0x0080
#define     Modal                   0x0100
#define     Floating                0x0200
#define     Workspace 0x0400
#define     Document                0x0800

window *    new_window(char *name, rectangle r, ushort flags);
void        set_winfns(window *w, winfn *close, winfn *size, winfn *draw);
void        set_winname(window *w, char *newname);
void        del_window(window *w);
void        show_window(window *w);
void        hide_window(window *w);
```

DESCRIPTION

A **window** is a bitmap as displayed on screen. A window's bitmap is guaranteed to have a zero origin, which corresponds with its top-left point on screen.

```
struct window { bitmap *b; rectangle r; ulong flags; short kind; void *data;
                winfn *close; winfn *resize; winfn *redraw; };
```

B points to the represented bitmap, and can be used for drawing. **R** holds the location of the window's bitmap on the screen in screen co-ordinates, while **flags** is a bit-field describing the window's

appearance and behaviour. The **kind** integer and the **data** pointer can be used by the programmer; they are initialised to zero and then ignored by the library.

The **winfns** are called by the window manager in response to certain user actions. The *close* function is called when the user attempts to close the window. If no such function exists, the default behaviour is for the window to be hidden. The *resize* function is called when the window is resized, and can be used to recalculate the window's appearance. The *redraw* function is called after the window has been resized or when a part of the window has been exposed, and should redraw the entire window. All of the winfns are passed a pointer to the affected window.

New_window creates and returns a pointer to a window with the given **name**. The rectangle **r** specifies where the window's bitmap rectangle appears on the screen, with zero being the top-left point of the screen. If an error occurs, *gerror* will be called, or the function will return NULL if *gerror* is NULL.

The **flags** argument is a bit-field. **Titlebar** gives the window a titlebar which can be used for moving it around the screen and also for displaying the window's name. **Closebox** gives the user a way of closing the window. **Maximize** gives the user a way of increasing the size of the window to its maximum, and **Resize** gives the user a method of changing the size of the window.

Attentive windows will receive the mouse click which activates them. Windows created with the **DoubleClicks** flag will produce mouse events which have the **DoubleClick** bit set in their kind field if the user clicks rapidly and repeatedly with the mouse buttons. **Modal** means the window will be in front of all other application windows when it is displayed, and no events will be sent to the other windows until it is hidden. **Floating** windows will appear in front of all other application windows even when not active.

A **Workspace** window can contain many **Document** windows. The appearance of these windows depends on the platform. Only one Workspace window can be created per application, but any number of Document windows can appear within it. On the Macintosh, a Workspace window will just be the screen and will have no window structure, but under Windows it is a normal window. Document windows automatically have the following flags set: Titlebar, Closebox, Maximize, Resize.

The **Buffered** flag is set by the library if the window requires the use of the gflush function for its contents to be drawn to the screen. The **Visible** flag is set by the library when the window is visible. New windows are invisible. The menubar, titlebars, resize boxes and window outlines are neither part of the screen bitmap nor a window's bitmap.

Set_winfns sets the functions to be called when the window is closed, resized or redrawn. *Set_winname* changes the name of the window as shown the window's titlebar. *Del_window* de-allocates the specified window, hiding it first if necessary. *Show_window* shows the specified window on the screen and ensures it is the frontmost application window. *Hide_window* causes the specified window to vanish from the screen.

Arithmetic functions

SYNOPSIS

```
#define     dx(r)       ((r).max.x-(r).min.x)
#define     dy(r)       ((r).max.y-(r).min.y)

point       pt(long x, long y);
rectangle   rect(long minx, long miny, long maxx, long maxy);
rectangle   rdiag(long minx, long miny, long width, long height);
rectangle   rpt(point min, point max);

point       addp(point p1, point p2);
point       subp(point p1, point p2);
point       mulp(point p, long i);
point       divp(point p, long i);
rectangle   raddp(rectangle r, point p);
rectangle   rsubp(rectangle r, point p);
rectangle   mulr(rectangle r, long i);
rectangle   divr(rectangle r, long i);
rectangle   insetr(rectangle r, long i);
rectangle   rcanon(rectangle r);
short       pinr(point p, rectangle r);
short       rxr(rectangle r1, rectangle r2);
short       eqp(point p1, point p2);
short       eqr(rectangle r1, rectangle r2);
short       rclip(rectangle *r1, rectangle r2);
```

DESCRIPTION

The functions *pt*, *rect*, *rdiag* and *rpt* construct geometrical data types from their components. The macros *dx* and *dy* give the width and height of a rectangle.

Addp returns the **point** sum of its arguments: *pt*(p.x+q.x, p.y+q.y). *Subp* returns the **point** difference of its arguments: *pt*(p.x-q.x, p.y-q.y). *Mulp* returns the **point** *pt*(p.x*a, p.y*a). *Divp* returns the **point** *pt*(p.x/a, p.y/a).

Raddp returns the **rectangle** *rpt*(*addp*(r.min, p), *addp*(r.max, p)); *rsubp* returns the **rectangle** *rpt*(*subp*(r.min, p), *subp*(r.max, p)). *Mulr* returns the **rectangle** *rpt*(*mulp*(r.min, a), *mulp*(r.max, a)); *Divr* returns the **rectangle** *rpt*(*divpt*(r.min, a), *divpt*(r.max, a)).

Insetr returns the **rectangle** *rect*(r.min.x+n, r.min.y+n, r.max.x-n, r.max.y-n).

Rcanon returns a **rectangle** with the same extent as **r**, canonicalized so that min.x <= max.x, and min.y <= max.y.

Pinr returns 1 if **p** is a point within **r**, and 0 otherwise. *Rxr* returns 1 if **r1** and **r2** share any point, and 0 otherwise. *Eqp* compares its argument **points** and returns 0 if unequal, 1 if equal. *Eqr* does the same for

its argument **rectangles**.

Rclip clips the **rectangle** pointed to by **r1** so that it is completely contained within **r2**. The return value is 1 if any part of ***r1** is within **r2**. Otherwise, the return value is 0 and ***r1** is unchanged.

Cursors & Fonts

SYNOPSIS

```
struct      cursor {
            point       offset;             /* bitmap offset from mouse location */
            uchar       white[2*16];        /* white mask */
            uchar       black[2*16];        /* black shape */
            void *      cp;                 /* library data: initialise to NULL */
};

struct      font {
            short       height;     /* height of a line */
            short       ascent;     /* top of bitmap to baseline */
            short       descent;    /* baseline to descender */
};

cursor *    get_cursor(char *name);
void        set_cursor(cursor *c);
font *      get_font(char *name, char *style, ushort size);
```

DESCRIPTION

A **cursor** is put in this structure:

```
struct cursor { point offset; uchar white[2*16]; uchar black[2*16]; void *cp; };
```

The arrays are to be arranged in rows, two characters per row, to give 16 rows of 16 bits each. A cursor is displayed on the screen by adding offset to the current mouse position, using **white** as a mask to white out the pixels where **white** is 1, and then setting all pixels to black where **black** is 1.

The *get_cursor* function finds a cursor with the given name in the application's resources and returns a pointer to it. If the named cursor cannot be found, the function will call *gerror*, or return NULL if gerror is NULL. The *set_cursor* function will change the application's cursor to the specified one. The **cp** field must be set to NULL initially if the cursor is from application data; it is used internally by the library.

A **font** is a typeface of a certain point size.

```
struct font {short height; short ascent; short descent; };
```

The **height** is the distance in pixels from the top of one line of text to the top of the next. The **ascent** and **descent** are respectively the distances above and below the font's baseline that the font characters actually extend.

The *get_font* function returns a pointer to a required font. A font has a **name**, a **style** and a **size** in points. The style is specified as a string containing space or comma separated words defining the style. An example style string might be "Bold, italic". The function returns NULL if the font cannot be

obtained.

Drawing functions

SYNOPSIS

```
typedef ulong           pixval;

void        bit_copy(bitmap *db, point p, bitmap *sb, rectangle r, pixval v);
void        texture_rect(bitmap *db, rectangle r, bitmap *sb, pixval v);
void        invert_rect(bitmap *db, rectangle r);
void        fill_rect(bitmap *db, rectangle r, pixval v);
void        draw_rect(bitmap *db, rectangle r, long w, pixval v);

void        draw_point(bitmap *db, point p, pixval v);
void        draw_line(bitmap *db, point p1, point p2, pixval v);
void        draw_arc(bitmap *db, point p0, point p1, point p2, pixval v);
void        fill_circle(bitmap *db, point p, long r, pixval v);
void        draw_circle(bitmap *db, point p, long r, pixval v);
void        fill_ellipse(bitmap *db, point p, long r1, long r2, pixval v);
void        draw_ellipse(bitmap *db, point p, long r1, long r2, pixval v);
point       draw_string(bitmap *db, point p, font *f, char *s, pixval v);
long        strwidth(font *f, char *s);
point       strsize(font *f, char *s);

/* Transfer code pixvals for drawing operations */

enum {
            Zeros       = 0x00,     DnorS       = 0x01,
            DandnotS    = 0x02,     notS        = 0x03,
            notDandS    = 0x04,     notD        = 0x05,
            DxorS       = 0x06,     DnandS      = 0x07,
            DandS       = 0x08,     DxnorS      = 0x09,
            D           = 0x0A,     DornotS     = 0x0B,
            S           = 0x0C,     notDorS     = 0x0D,
            DorS        = 0x0E,     Ones        = 0x0F
}

/* Colour pixvals */

#define     BLACK       0x00000000L
#define     WHITE       0xFFFFFF00L
#define     BLUE        0x0000FF00L
#define     YELLOW      0xFFFF0000L
#define     GREEN       0x00FF0000L
#define     MAGENTA     0xFF00FF00L
#define     RED         0xFF000000L
#define     CYAN        0x00FFFF00L
#define     GREY        0x7F7F7F00L
#define     LTGREY      0xBFBFBF00L
```

#define DKGREY 0x3F3F3F00L

DESCRIPTION

All of the drawing operations draw into a destination bitmap **db**. Some operations transfer pixel values from a source bitmap **sb**, while most take pixel values given in a **pixval** argument **v**. A **pixval** is a data type which has two purposes: to specify how to compute each destination pixel as a function of source and destination pixels; and to supply a source pixel value for those drawing operations that do not take source pixel values from another bitmap.

The high three bytes of a **pixval** is used to specify the source colour, using the red-green-blue scheme. The highest byte encodes the intensity of red light, the second highest byte encodes the intensity of green light, and the third highest byte encodes the intensity of blue light.

The lowest byte of a **pixval** holds a transfer code. The sixteen transfer code pixvals give all possible bitwise operations of the source **S** and destination **D**. For the purposes of these bitwise operations, black will always have a pixel value equal to all zeros, while white will have a pixel value which is all ones. If a source colour is specified but the transfer code is left as zero, the transfer code **S** is assumed (which just has the effect of copying the source colour into the destination bitmap).

Black and white can always be represented in a bitmap, while other colours may have to be approximated. For bitmaps of depth 1, all colours except grey are mapped to either black or white as follows: hues near blue, red and magenta will map to solid black, while lighter hues near green, cyan and yellow will map to solid white. The grey pixval will result in dithering to produce a grey effect. This will work with text and all line drawing, but may not always give desirable results when drawing thin lines. For depth 2, colours will still map to black or white, but grey will map to a solid grey. For depths greater than 2, better approximations are used.

Bit_copy takes bits from rectangle **r** in the source bitmap **sb**, and overlays them on a congruent rectangle with the min corner at point **p** in the destination bitmap **db**. The **v** parameter is used to specify how source and destination bitmap pixels are combined, so that its colour component is ignored.

If the source and destination bitmaps have different depths, the source rectangle is first converted to have the same depth as the destination. All of the drawing graphics functions clip the rectangle against the source and destination bitmaps, so that only pixels within the destination bitmap are changed, and none are changed that would have come from areas outside the source bitmap.

Texture_rect fills the rectangle **r** in the destination bitmap **db** with copies of the source bitmap **sb**. The copies are aligned so that repeating patterns will look correct. It also ignores the colour component of **v**, but uses it as a transfer code. *Invert_rect* inverts black and white in the required rectangle. Inversion of coloured areas is not guaranteed to produce the desired results.

Fill_rect fills a rectangle with a certain colour, specified as a pixval **v**. *Draw_rect* draws four lines of width **w** within the given rectangle. If **w** is negative, *draw_rect* draws the lines outside the given rectangle. The lines will be of a colour and transfer code specified by **v**.

Draw_point changes the value of the destination point **p** in bitmap **db** to the required pixel value **v**. *Draw_line* draws a line segment in bitmap **db** with a pixel value **v** from point **p1** to **p2**. The segment is

half-open: **p1** is the first point of the segment and **p2** is the first point beyond the segment, so adjacent segments sharing endpoints abut.

Draw_arc draws a circular arc centred on **p0**, travelling anti-clockwise from **p1** to **p2**, or to a point on the line passing through **p0** and **p2**. The arc will be one pixel thick and of pixel value **v**. *Fill_circle* fills a circle centred on **p** with radius **r** using the pixel value **v**. *Draw_circle* draws a one pixel thick circle of the given pixel value instead of filling it. *Fill_ellipse* and *draw_ellipse* are similar, except the horizontal semi-axis is **r1** and vertical semi-axis is **r2**.

Draw_string draws the text characters given by the null-terminated string **s** into bitmap **db**, using font **f** and pixel value **v**. The upper left corner of the first character (i.e., a point that is **f->ascent** above the baseline) is placed at point **p**, and subsequent characters are placed on the same baseline, displaced to the right by the previous character's width. *Draw_string* returns the point in the destination bitmap after the final character of **s** (or where the final character would be drawn, assuming no clipping; the returned value might be outside the destination bitmap).

The bounding box for text to be drawn with *draw_string* in font **f** can be found with *strsize*; it returns the **max** point of the bounding box, assuming a **min** point of (0,0). *Strwidth* returns the x-component of the **max** point.

Menus

SYNOPSIS

```
typedef void voidfn(void);          /* function which takes no arguments and returns nothing */

struct      menuitem {
            char *      name;       /* name of menu item, NULL=end of list */
            ushort      key;        /* key equivalent, 0=none */
            short *     state;      /* pointer to state variable */
            voidfn *    action;     /* action to perform when item is chosen */
};

typedef menuitem *menu;             /* array of menuitem pointers, NULL terminated */

#define     Disabled    0x00
#define     Enabled     0x01
#define     Ticked      0x02

extern short menu_item;             /* array index of last selected item */
```

DESCRIPTION

A **menuitem** is a single item in a menu. A **menu** is an array of menuitems terminated by an empty menuitem.

```
struct menuitem { char *name; short key; char *state; voidfn *action; };
typedef menuitem *menu;
```

The first menuitem in a menu designates the name of the entire menu. Each menuitem has a **name**, and can have a shortcut **key**. Holding down Control or Command while pressing this key has the effect of selecting that menuitem.

The appearance of the menuitem is determined by a state variable pointed to by **state**. The item can be **Enabled** or **Disabled** (grey), and may optionally be **Ticked**. If **state** is NULL, the item is disabled. The action function is called when the item is selected. The library global **menu_item** is set to the array index of the selected item when action is called. The third item in a menu after the menu's name will thus cause **menu_item** to be set to three if it is selected.

Every application can have a **menubar**. A menubar is a NULL terminated array of menus. On Macintosh platforms, the menubar will appear in its normal place at the top of the screen. On Windows and X-Windows platforms, the menubar will appear in the first window the program creates. The menubar is passed as an argument to *ginit*.

If a menu's first item (the menu's name) is disabled, all of the items in that menu will be disabled, except the name itself. This provides a way of de-activating an entire menu without modifying the state of each item separately.

The menuitem states can be modified during the *adjust_menus* function, which is one of the arguments to *ginit*. The *adjust_menus* function is called just before any menus are actually displayed, and can ensure that the current state of the menus matches the program's state.

Mouse Events

SYNOPSIS

```
struct      mouse {
            uchar       kind;       /* mouse event kind bit array */
            uchar       buttons;    /* mouse button state bit array: LMR=124 */
            point       xy;         /* location of mouse */
};

#define     NoButton                0x00
#define     LeftButton              0x01
#define     MiddleButton            0x02
#define     RightButton             0x04

#define     MouseMove               0x00
#define     MouseDown               0x10
#define     MouseUp                 0x20
#define     MouseTimer              0x40
#define     DoubleClick             0x80

#define     LeftButtonDown          (MouseDown | LeftButton)
#define     MiddleButtonDown        (MouseDown | MiddleButton)
#define     RightButtonDown         (MouseDown | RightButton)
#define     LeftButtonUp            (MouseUp | LeftButton)
#define     MiddleButtonUp          (MouseUp | MiddleButton)
#define     RightButtonUp           (MouseUp | RightButton)
```

DESCRIPTION

A **mouse** structure contains information about the mouse's location, button state and activities:

```
struct mouse { uchar kind; uchar buttons; point xy };
```

Kind is a bit field which reports what kind of mouse event has just occurred. If the mouse was just moved, kind will be set to **MouseMove**. If the mouse timer mechanism (see below) has just timed out the kind will be **MouseTimer**. If one of the mouse buttons has been pressed or released the kind field will be set to a combination of bits.

Which button was pressed or released is recorded in the low four bits of the kind field; these low four bits will be one of **LeftButton**, **MiddleButton** or **RightButton**. If a button was pressed, the kind field will have the **MouseDown** bit set. If a button was released the **MouseUp** bit will be set. If the window from which the mouse event came has the **DoubleClicks** flag set, a rapid repeated pressing of a mouse button will cause the second **MouseDown** event to also have the **DoubleClick** bit set in its kind field. The definitions **LeftButtonDown** through to **RightButtonUp** are for convenience only.

Buttons is bit field which describes the current state of the mouse buttons; **buttons&LeftButton** is set when the left mouse button is depressed, **buttons&MiddleButton** when the middle button is

depressed, and **buttons&RightButton** when the right button is depressed.

For mouses with less that three buttons, the middle and right buttons can be simulated. Holding down the Control or Option key and pressing a mouse button is the same as depressing the middle button, while holding down the Shift key and pressing a mouse button is the same as depressing the right button.

The mouse position is found in the point **xy**. This location will be relative to the window for which the event was generated. A window's top left point is (0,0).

Events

SYNOPSIS

```
short     start_timer(long msec);
short     set_mouse_delay(long msec);
void      delay(long msec);
short     can_event(void);
short     can_timer(void);
short     can_mouse(window *w);
short     can_key(window *w);
long      get_timer(void);
mouse     get_mouse(window *w);
ushort    get_key(window *w);
void      unget_mouse(window *w, mouse m);
void      unget_key(window *w, ushort k);
```

DESCRIPTION

There are two main sources of events for an application: events that are window based and events that are application based. Keyboard and mouse events occur within the context of a window, while timer, system and inter-application events are on an application-level.

All events are stored in a queue of their own and retrieved using the *get* event functions. The *get* event functions will all block until there is an event of the required type. The *can* functions return a non zero result when the relevent queue is not empty, and zero when it is empty. The *unget* event functions put an event back on to the start of a queue.

The *can_event* function returns a non-zero result if there are any events for the application to handle. It also gives some processor time to the library to allow menu handling and window updating to occur. All of the *can* functions have this quality, and so they should be called before the *get* functions which can block processing.

The application can have one timer, which sends events to the application at regular intervals. The *start_timer* function starts this timer with an interval of **msec** milliseconds and returns a non-zero value if successful. If the timer could not be started, it will call *gerror* or return zero if *gerror* is NULL. The timer can be halted by calling *start_timer* with an **msec** value of zero. The *can_timer* function returns a non-zero result if there is at least one timer event in the event queue. The *get_timer* function dequeues one queued timer event, or waits until there is one before dequeuing it.

The *delay* function suspends the application for the required number of milliseconds. This should only be used for short intervals (less than a second). Longer intervals should use the timer mechanism, which is non-blocking and allows background processing.

The library associates the keyboard and mouse events with the windows for which they are relevent. The functions which handle these events thus take a pointer to the window of interest as their first argument. Only a certain number of the latest events will remain queued for a window.

Characters typed on the keyboard will be sent to the currently active window. *Can_key* returns a non-zero result if there are keyboard events queued for the window, and zero otherwise. *Get_key* dequeues one character from the keyboard event queue of a window, or waits until there is one. *Unget_key* puts a character back on to the window's keyboard event queue.

The characters returned from *get_key* will be normal ASCII chars unless they are the result of typing with certain special keys. The function keys, arrow keys and editing keys (insert, delete, home, end, page up and page down) all generate special key codes. These key codes can be found in the include file "stdkey.h". The escape, tab and enter/return keys generate normal ASCII escape, horizontal tab and newline codes.

When the mouse moves or a mouse button is depressed or released in the active window, a new mouse event is queued for that window by the event mechanism. Mouse clicks in inactive windows which have the **Attentive** bit set will also be queued.

Mouse events can also be generated if a mouse button is held down for longer than a certain time. The *set_mouse_delay* function sets this time interval to be **msec** milliseconds. If a mouse button is held down for longer than this time, the latest mouse event will be repeated in the queue automatically, and this will continue to happen at the same time interval until the mouse button is released. The effect is similar to the way keyboard keys repeat when held down. Initially this feature is not active, and it can be de-activated by called *set_mouse_delay* with **msec** equal to zero.

Can_mouse returns a non-zero result if there are mouse events queued for the window, and zero otherwise. *Get_mouse* dequeues one mouse structure from the mouse event queue of a window, or waits until there is one. *Unget_mouse* puts a mouse structure back on to the window's mouse event queue.

A mouse structure looks like:

```
struct mouse { uchar kind; uchar buttons; point xy; };
```

Kind is a bit field which report what kind of mouse event has just occurred. **Buttons** is a bit field which will reflect the current state of the mouse's buttons. The current mouse position is found in **xy**, and this will be in the co-ordinate system of the window.

There are other types of events which interact with applications differently. Window manager events cause the library to call functions related to each window. The *close*, *resize* and *redraw* functions are called by the library in response to such events. If the *close* function does not exist for the affected window, the library merely hides it by default. The *resize* function occurs before the *redraw* function, and both occur after the window manager has changed the size of a window. *Redraw* can also occur when a part of a window has been exposed.

Menu events are handled entirely by the library too. Once a **menubar** has been set up, interaction with it is through changing **menuitem** states (disabling, enabling or ticking items). This can be done as an integral part of a program, or in a single *adjust_menus* function as specified in the *ginit* argument.